MULT - A Multitasking Ultracomputer Language with Timing, I & II

Allan Gottlieb and Clyde Kruskal

Courant Institute
251 Mercer Street
New York, NY 10012

Ultracomputer Note #15

28 December 1980

1. Introduction

[UC] introduced the idea of an ultracomputer, a parallel processor based on the "shuffle" connections [St]; reviewed various basic algorithms for such an ensemble of processors; and analyzed their asymptotic time complexities. However, one will want to know exact, rather than merely asymptotic, efficiencies when designing and using an ultracomputer. Moreover, some algorithms are difficult to analyze analytically (e.g. linear programming and backtracking) and only actual practice gives us a feeling for their efficiencies. The MULT (*M*ultiprocessor *U*ltracomputer *L*anguage with *T*iming) system described in this note is an attempt to deal with these issues by timing simulated executions of ultracomputer programs.

The MULT system includes a PL/I [LRM] based programming language and a simulator for running and timing MULT language programs. Although PLUS, an ultracomputer programming language and simulator has previously been designed and implemented [UCN10] and [UCN14], that system ignores timing issues. MULT, on the other hand, is expressly designed for gathering timing information and willingly sacrifices some programming ease to do so. Interprocessor communication is modeled in MULT by message passing; each message must be explicitly sent and received.

Due to the modular design of MULT, only a minor effort is needed to reconfigure it to support other interconnection schemes. In particular, the layered ultracomputer [UC] and the multidimensional ultracomputer variants of Harrison and Kalos (see [UCN6]) are easy to implement.

2. User's Manual

As suggested in [UCN3], we assume that each processor in the ultracomputer will execute the same program. Note that this does not imply a SIMD architecture since the processors execute asynchronously and conditional statements (with condition depending on processor number) are permitted. Our basic

idea is to write such programs as PL/I procedures containing an additional parameter representing the processor number. Then this procedure is invoked asynchronously, once for each processor. The PL/I multitasking facility allows the multiple invocations of the procedure thereby created to run "in parallel". Each such invocation is called a *task*.

Thus the MULT language is merely an extension of PL/I. The language extensions involve primarily timing and message passing, although some other generally useful features have been added. The system is easy to use, and the reader is encouraged to look at the example in section III, which illustrates its use.

2.1. Accessing the Macro Package

Every procedure that sends or receives messages must include the MULT macro package for the relevant sized ultracomputer (or have an outer procedure that includes this package). This is done by writing the statement

```
%INCLUDE MULTn;
```

inside the procedure, where n is the number of processors. For example to simulate an 8 processor ultracomputer one writes

```
%INCLUDE MULT8;
```

The macro package defines constants (e.g. TRUE and FALSE), read-only arrays (e.g. SHUFFLE), generally useful macros (e.g. EVEN and ODD), and special macros for message handling and program timimg. In addition, MULT declares (and uses) some PL/I builtin functions. All these items are described in additional detail below.

2.2. Setting Processor Numbers

When sending and receiving messages and updating elapsed time, each task must supply the number of the processor which it is simulating. Since this parameter is used in many macros and tends not to vary, MULT permits one to write

```
PROCESSOR_NUMBER_IS (P#);
```

(where P# is a variable used to represent the processor number) once and omit the processor number parameter from the relevant macros.

2.3. Timing

A major objective of MULT is to determine the time each processor spends waiting for messages. To do this MULT maintains (synchronized) clocks for all processors and MULT time stamps each message. These clocks, initialized to zero, are advanced by both computation and message passing.

2.3.1. Computation Time Ideally, the simulator should determine computation times automatically, but unfortunately, user intervention is required. Each task must periodically calculate an estimated elapsed time DELTA_T (the incremental computation time used) and transmit this value to the simulator by

[illegible]

invoking the macro

UPDATE_TIME (DELTA_T, P#);

In order for MULT to time stamp messages and calculate processor waiting times, it is necessary that a processor's clock be accurate whenever that processor sends or receives a message; however, the clocks need not be correct during computation. Thus one UPDATE_TIME macro may be used to transmit the time required for an uninterupted sequence of computations. The reader is advised to inspect the summing example in section III where we have adopted the conventions of updating time prior to computation and charging a subroutine for its CALL and RETURN (this latter convention is justified in [Kn]).

2.3.2. Message Passing Time Time is required to send a message, to transmit it across the "wire", and to recieve it. These times, any (or all) of which may be zero, are denoted by s, w, and r respectively, and are specified at the beginning of each simulation.

Suppose that processor p sends a message to processor q at time t (according to p's clock). Processor p's clock is advanced to $t+s$ and the message is time stamped $t+s+w$ (the time it will arrive at q). Suppose further that q requests receiving this message at time t' (according to q's clock). If $t' \geq t+s+w$ (the time stamp), the message has already arrived and q's clock is advanced to $t'+r$; if $t' < t+s+w$, q must wait $t+s+w-t'$ time units for the message to arrive. MULT records the waiting time and advances q's clock to $t+s+w+r$.

2.4. Message Passing

Each processor in an ultracomputer is directly connected to four neighbors – referred to as its right, left, shuffle, and unshuffle neighbors (see [UC]). Moreover, we consider processors 2i and 2i+1 as "partners". These neighbor and partner connections are reflected in the ten basic message passing macros:

```
SEND_RIGHT (M, P#);
SEND_LEFT (M, P#);
SEND_SHUFFLE (M, P#);
SEND_UNSHUFFLE (M, P#);
SEND_PARTNER (M, P#);
RECEIVE_RIGHT (M, P#);
RECEIVE_LEFT (M, P#);
RECEIVE_SHUFFLE (M, P#);
RECEIVE_UNSHUFFLE (M, P#);
RECEIVE_PARTNER (M, P#);
```

where M is the message to be sent, and P# is the processor number. If the processor numbers have been set, the final parameter P# may be omitted. Each of the send macros indicates that the message M is to be sent to the neighbor denoted in the macro name, and each of the receive macros indicates that a message is to be received from the neighbor denoted in the macro name and stored in

M.

For example, with an 8-processor ultracomputer,

SEND_RIGHT('HELLO', 3);

signifies that processor 3 is sending the message 'HELLO' to processor 4. Processor 4 should then invoke the corresponding macro

RECEIVE_LEFT(MSG, 4);

This macro stores the message 'HELLO' in processor 4's variable MSG. If the processor numbers have been set, these macro calls can be abbreviated as

SEND_RIGHT('HELLO');

and RECEIVE_LEFT(MSG);

respectively.

Ultracomputer algorithms often contain global "shuffles" (similarly "unshuffles") where each processor sends to its shuffle neighbor. Therefore MULT supplies

SHUFFLE(M, P#);

and UNSHUFFLE(M, P#);

each of which expands into an appropriate SEND and RECEIVE. If the processor numbers have been set, these two macros can be abbreviated as

SHUFFLE(M);

and UNSHUFFLE(M);

respectively.

2.5. Miscellaneous Constructs

The macro package defines the following generally useful constructs:

Constants Defined by the Macro Package

#P	=	the number of processors
MAX_P#	=	#P - 1
LOG_#P	=	$\log_2$(#P)
TRUE	=	'1'B
FALSE	=	'0'B

Read Only Arrays Used by the Macro Package

RIGHT_PE(0 : MAX_P#)
LEFT_PE(0 : MAX_P#)
SHUFFLE_PE(0 : MAX_P#)
UNSHUFFLE_PE(0 : MAX_P#)
PARTNER_PE(0 : MAX_P#)

These arrays model the ultracomputer connections, where PE abbreviates processing element. For example, on an 8-processor ultracomputer RIGHT_PE(3) = 4, LEFT_PE(3) = 2, SHUFFLE_PE(3) = 6, UNSHUFFLE_PE(3) = 5, and PARTNER_PE(3) = 2. These arrays are read-only in a moral sense: Although the user can modify them, he should not. They should be treated as if they were

functions.

Functions Defined by the Macro Package

EVEN(N) returns TRUE if N is even.

ODD(N) returns not EVEN(N).

LOW_BITS(P#,J) returns the bit string consisting of the J low-order bits of P#.

HIGH_BITS(P#,J) returns the bit string consisting of the J high-order bits of the LOG_#P-bit number P#.

BIT_ON(P#,J) returns TRUE if the j^{th} low order bit of P#, i.e. the bit corresponding to 2^{J-1} is a 1.

BIT_OFF(P#,J) returns not BIT_ON(P#,J).

Other Reserved Words

All names mentioned above are reserved words and should not be redeclared or redefined. The following PL/I builtin functions are also used in the macro package and declared therein:

LENGTH
SUBSTR
BIT
MOD
COMPLETION

A few other names are used by the macro package for internal communication. All such names start with "$$" and are reserved.

3. Example

3.1. Summing

For completeness we excerpt the following description of the summing algorithm from [UC].

(a) Replace w_n by $w_{n-1} + w_n$ for each odd n.[1]

(b) Proceeding recursively, apply summing to the odd elements. (This can be done by first unshuffling then applying summing to the upper half of the processors, then shuffling.) At the end of this step, every odd processor p_n will contain

[1]The actual version in [UC] is for any associative operator ■.

$w_0 + \ldots + w_n$.

(c) Replace w_n by $w_{n-1} + w_n$ for each even $n>0$.

The following MULT program is an 8-processor ultracomputer implementation of summing. Thus, the program is to be called asynchronously 8 times with N = 0,1,...,7. The parameter LB in the inner procedure is used to implement the recursion appearing in step (b) above. At each level of the recursion, the only processors "active" are those numbered LB, LB+1, ..., 7.

```
SUMMING: PROC(W, N) OPTIONS (REENTRANT);
   %INCLUDE MULT8;
   PROCESSOR_NUMBER_IS(N);
   DCL (W, N) FIXED BIN;
   UPDATE_TIME(8);

   CALL SUMMING1(0);

   SUMMING1: PROC(LB) RECURSIVE OPTIONS (REENTRANT);
      DCL LB FIXED BIN;
      DCL W_LEFT FIXED BIN;
      UPDATE_TIME(8);

      UPDATE_TIME(1);
      IF  LB <= N  THEN DO;
          UPDATE_TIME(1);
          IF  EVEN(N)  THEN
             SEND_RIGHT(W);
          ELSE DO;
             RECEIVE_RIGHT(W_LEFT);
             UPDATE_TIME(1);  W = W_LEFT + W;
          END;
       END;

      UPDATE_TIME(1);
      IF  LB+1 < MAX_P#   THEN DO;
         UNSHUFFLE(W);
         CALL SUMMING1( (LB + #P)/2 );
         SHUFFLE(W);
      END;

      UPDATE_TIME(2);
A:    IF  LB<N & N<MAX_P#   THEN DO;
         UPDATE_TIME(1);
        IF  ODD(N)  THEN
           SEND_RIGHT(W);
        ELSE DO;
           RECEIVE_LEFT(W_LEFT);
           UPDATE_TIME(1);  W = W_LEFT + W;
        END;
       END;

       UPDATE_TIME(5);
   END SUMMING1;

     UPDATE_TIME(5);
END SUMMING;
```

Here somewhat arbitrary DELTA_T values have been used: Each PL/I statement is given a cost of 1 unit except for line A which is given a cost of 2 units since it is slightly more complicated than the others. Each procedure is charged 8 units as the cost for its invocation and 5 units as the cost for its return. We use the convention that each procedure pays for both its own invocation and return.

For the summing example – with s, w, and r each equal to 1 – MULT produces the following timing report:

```
P#   TOTAL   COMPUTING   WAITING   S   R   SEND   RECEIVE
 0     78       65           4     5   4     5       4
 1     85       67           8     5   5     5       5
 2     88       67          11     5   5     5       5
 3     92       67          15     5   5     5       5
 4     95       68          16     6   5     6       5
 5     95       70          13     6   6     6       6
 6     98       71          14     7   6     7       6
 7     91       70          10     4   7     4       7
```

In this report the columns have the following meanings:

P# – The number of the processor.

TOTAL – The number of elapsed time units until this processor completed execution.

COMPUTING – The number of time units this processor spent computing (i.e. the sum of the DELTA_Ts).

WAITING – The number of time units this processor was idle (i.e. the time spent waiting for messages).

S – The number of time units spent sending messages (i.e. s times the number of messages sent).

R – The number of time units spend receiving messages (i.e. r times the number of messages received).

SEND – The number of messages sent.

RECIEVE – The number of messages received.

Acknowledgements The authors wish to thank Jay Klein who was largely responsible for the implementation. In addition, we thank the entire NYU ultracomputer group for many helpful discussions.

4. References

[Kn] Donald Knuth, "Estimating the Efficiency of Backtrack Programs", *Math. of Computation* 29, pp. 121-136, 1975.

[LRM] *OS PL/I Checkout and Optimizing Compilers: Language Reference Manual,* GC33-0009-4, IBM, 1976.

[St] Harold S. Stone, "Parallel Processing with the Perfect Shuffle", *IEEE Trans.* C-20, pp. 153-161, 1971.

[UC] J. T. Schwartz, "Ultracomputers", *ACM TOPLAS* 2, pp. 484-521, 1980.

[UCN3] J. T. Schwartz, "Preliminary Thoughts on Ultracomputer Programming Style", Ultracomputer Note #3, NYU, 1979.

[UCN6] Clyde Kruskal and Larry Rudolph, "Observations Concerning Multidimensional Ultracomputers", Ultracomputer Note #6, NYU, 1980.

[UCN10] Allan Gottlieb, "PLUS: A PL/I Based Ultracomputer Simulator, I", Ultracomputer Note #10, NYU, 1980.

[UCN14] Allan Gottlieb, "PLUS: A PLI Based Ultracomputer Simulator, II", Ultracomputer Note #14, NYU, 1980.

Table of Contents